ON CUDA IMPLEMENTATION OF A MULTICHANNEL ROOM IMPULSE RESPONSE RESHAPING ALGORITHM BASED ON P-NORM OPTIMIZATION

Radoslaw Mazur, Jan Ole Jungmann, and Alfred Mertins

Institute for Signal Processing
University of Lübeck
23562 Lübeck, Germany
{mazur,jungmann,mertins}@isip.uni-luebeck.de

ABSTRACT

By using room impulse response shortening and shaping it is possible to reduce the reverberation effects and therefore improve speech intelligibility. This may be achieved by a prefilter that modifies the overall impulse response to have a stronger attenuation. For achieving a spatial robustness, multichannel approaches have been proposed. Unfortunately, these approaches suffer from a very high computational cost and are far too slow for being of practical use in applications where filters have to be designed in real-time. In this work we tackle this drawback using a CUDA implementation and achieve a speedup of over 130 times.

Index Terms— Optimization, reshaping, shortening, room impulse response (RIR), CUDA

1. INTRODUCTION

Room impulse response (RIR) shaping is a method for reducing the perceived echoes in acoustic scenes. It may be performed using a prefilter that modifies the overall impulse response. It is not necessary to invert the channel and get the exact signal [1, 2]. As proposed in [3, 4] it is sufficient to shape the overall RIR with respect to the human auditory system. Therefore, in order to reduce the reverberation, it is sufficient to equalize the RIR in a such way that the audible echoes are removed, while the inaudible ones may stay unaffected. As a further benefit, this approach lightens the pressure of designing the prefilter [5].

The approach in [4] takes advantage of the fact that echoes may remain in the signal and are unperceivable if they fall below the temporal masking curve of the human auditory system. Although the exact temporal masking curve is signal dependent [6], an average signal-independent masking curve has been found in [7] that is triggered by the direct sound and was used in [4] to design the prefilter.

The methods for prefilter design in [3, 4, 5] are gradient descent approaches, which minimize either an Euclidean norm, an ℓ_p-norm, the ∞-norm or a combination of them. Depending on the used norm, these algorithms have different computational costs, but usually this is not a problem using a modern hardware.

This single-channel approach from [4] is not spatially robust in the sense that already small movements of speaker or microphones may results in substantially changed RIRs and a reduced overall performance. Based on the spatial sampling principle [8] a multichannel extension has been proposed in [9]. It allows for an equalization of multiple points in space by using several loudspeakers and microphones. When the sampling points are chosen according to the spatial sampling principle an entire area can be equalized. This approach is much more robust, as it allows for movements of the listener in this whole area.

Unfortunately, the multichannel approach is computationally very demanding. Even for a mid-size configuration with 13 loudspeakers and 57 microphones, as proposed in [9], several hours of computation are needed before the ℓ_p-norm based gradient descent converges. In this work we propose to use the computational power of modern, CUDA enabled graphics hardware [10]. By carefully designing the CUDA programs an over 130 times speedup will be achieved, and the computation time can be reduced to only few minutes or even seconds. When going to smaller sized configurations, a near real-time processing will be possible.

2. PROBLEM STATEMENT

In a multichannel RIR reshaping setup, N_m microphones in the listening area are used. The source signals are played using N_s loudspeakers. The individual RIRs with length L_c from loudspeaker k to microphone i are denoted as $c_{ik}(n)$. With $h_k(n)$ being a length-L_h prefilter to the k-th loudspeaker, the overall RIR to the i-th microphone is given by

$$g_i(n) = \sum_{k=1}^{N_s} h_k(n) * c_{ik}(n) = \sum_{k=1}^{N_s} \mathbf{C}_{ik}\mathbf{h}_k \tag{1}$$

with $\mathbf{C}_{ik}$ being an $L_g \times L_h$ convolution matrix of $c_{ik}(n)$. The length of the overall RIRs $g_i(n)$ is $L_g = L_c + L_h - 1$.

The task of filter shaping in this multichannel setup is to design the prefilters $h_k(n)$ in such a way that all overall RIRs $g_i(n)$ have reduced perceived echoes. For achieving this, we define the unwanted part of the reshaped RIRs as

$$g_{ui}(n) = w_{ui}(n)g_i(n) \tag{2}$$

with $w_{ui}(n)$ being a window function with the length L_g designed according to the average temporal masking curve from [7]. The windows $w_{ui}(n)$ usually have the same basic shape, with their nonzero parts starting 4 ms after the main peaks of the corresponding impulse responses $g_i(n)$.

In addition, we also define the desired parts

$$g_{di}(n) = w_{di}(n)g_i(n) \tag{3}$$

This work has been supported by the German Research Foundation under Grant No. ME1170/3-1.

of the overall responses, with windows $w_{di}(n)$ being nonzero (and constant) from the main peaks of $g_i(n)$ until 4 ms after that. The task is now to minimize some function of $|g_{ui}(n)|$ while keeping $|g_{di}(n)|$ constant. Here we follow the approach from [4] using the p-norm, as it allows for a shaping without major changes of the frequency responses of filters $g_i(n)$. The optimization problem is given by

$$\mathrm{MIN}_h : f(\mathbf{h}) = \log\left(\frac{f_u(\mathbf{h})}{f_d(\mathbf{h})}\right) \tag{4}$$

with

$$f_u(\mathbf{h}) = ||\mathbf{g}_u||_{p_u} = \left(\sum_{i=1}^{N_m}\sum_{k=0}^{L_g-1}|g_{ui}(k)|^{p_u}\right)^{\frac{1}{p_u}} \tag{5}$$

and

$$f_d(\mathbf{h}) = ||\mathbf{g}_d||_{p_d} = \left(\sum_{i=1}^{N_m}\sum_{k=0}^{L_g-1}|g_{di}(k)|^{p_d}\right)^{\frac{1}{p_d}} \tag{6}$$

where $\mathbf{h} = [\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_{N_s}]$, $\mathbf{g}_u = [\mathbf{g}_{u1}, \mathbf{g}_{u2}, \ldots, \mathbf{g}_{uN_m}]$, and $\mathbf{g}_d = [\mathbf{g}_{d1}, \mathbf{g}_{d2}, \ldots, \mathbf{g}_{dN_m}]$. For the minimization of (4) we use a gradient descent approach with the learning rule

$$\mathbf{h}^{l+1} = \mathbf{h}^l - \mu(l)\nabla_{\mathbf{h}} f(\mathbf{h}^l) \tag{7}$$

with

$$\nabla_{\mathbf{h}} f(\mathbf{h}) = \left[\sum_{k=1}^{N_m}\mathbf{C}_{k1}^T\boldsymbol{\zeta}_k, \sum_{k=1}^{N_m}\mathbf{C}_{k2}^T\boldsymbol{\zeta}_k, \ldots, \sum_{k=1}^{N_m}\mathbf{C}_{kN_s}^T\boldsymbol{\zeta}_k\right] \tag{8}$$

and

$$\boldsymbol{\zeta}_k = \frac{1}{f_u(\mathbf{h})^{p_u}}\mathbf{b}_{uk} - \frac{1}{f_d(\mathbf{h})^{p_d}}\mathbf{b}_{dk}, \tag{9}$$

where

$$b_{uk}(n) = w_{uk}(n)\cdot \mathrm{sgn}(g_{uk}(n))\cdot|g_{uk}(n)|^{p_u-1}, \tag{10}$$

$$b_{dk}(n) = w_{dk}(n)\cdot \mathrm{sgn}(g_{dk}(n))\cdot|g_{dk}(n)|^{p_d-1}, \tag{11}$$

and $\mu(l)$ is an adaptive step-size parameter. Here, we use $p_u = 20$ and $p_d = 10$ as proposed in [4].

Using the gradient descent, as in (7), all prefilters are calculated in dependency of all overall RIRs $g_i(n)$ simultaneously. This assures an equalization of all filters and, according to the spatial sampling principle, the whole listening area is equalized. Unfortunately, for achieving convergence the step size $\mu(l)$ in (7) has to be very small in the vicinity of the optimum, and therefore a very huge number of iterations is needed. This means that even when dealing with mid-sized configurations, the computational costs are too high for the algorithm being useful for a quick design of optimal prefilters.

Therefore we propose an implementation using the CUDA framework on modern graphics hardware. But before we present this CUDA implementation, we first discuss the traditional approach using a CPU and identify the computational complexity which slows down the whole process.

3. CPU/MATLAB IMPLEMENTATION

The whole algorithm can be separated into four parts: (a) Calculation of the overall RIRs $g_i(n)$ using (1). (b) Calculation of $\boldsymbol{\zeta}_k$ using (9), which is actually the main part of the algorithm. (c) Using $\boldsymbol{\zeta}_k$ to estimate the gradient $\nabla_{\mathbf{h}} f(\mathbf{h})$ using (8). (d) Estimating the step size $\mu(l)$ and making the gradient descent using (7). The timings of these parts are given in Table 1. Using an Intel Xeon CPU with 3Ghz and overdrive technique, a highly optimized Matlab implementation needs roughly 350 ms for a single iteration of the previously mentioned (13×57)-system with $L_c = 2000$ and $L_h = 3000$. As for convergence usually 10^4 to 10^5 iterations are needed, the whole process is too slow for being of practical use.

Table 1: Comparison of the duration of the calculation of the single steps of one iteration of the algorithm using an implementation in Matlab. Problem size: 13×57. Times are given in ms.

Calculation	Time	%	Step	Time	%
(a) Eq. (1)	144.99	41.3	FFT	1.45	0.4
			M+A	136.66	38.9
			IFFT	6.88	2.0
(b) Eq. (9)	57.51	16.4		57.51	16.4
(c) Eq. (8)	148.12	42.2	FFT	6.88	2.0
			M+A	139.79	39.8
			IFFT	1.45	0.4
(d) Eq. (7)	0.20	0.1		0.20	0.1
Total	**350.82**	100.0			

In the following we go through the single calculation steps and identify the time consuming parts.

The calculation of $g_i(n)$ using (1) in part (a) consists of a summation of multiple convolutions. This can be performed efficiently in the frequency domain, where the convolution becomes a multiplication. As the RIRs $c_{ik}(n)$ do not change, they may be transformed in advance using the FFT: $\boldsymbol{C}_{ik} = \mathrm{FFT}(\boldsymbol{c}_{ik}, L_n)$ with $L_n > L_g$ being the FFT size. For a reduced number of FFTs the additions may also be performed in the frequency domain. The rule reads

$$\mathbf{g}_i = \mathrm{Re}\left(\mathrm{IFFT}\left[\sum_{k=1}^{N_s}\boldsymbol{C}_{ik}\odot\mathrm{FFT}[\mathbf{h}_k, L_n]\right]\right) \tag{12}$$

with $\odot$ being a point-wise multiplication of two vectors. With caching of the intermediate results, only N_s+N_m FFTs are needed. (Here, we assume the complexity of FFT and IFFT is the same.) Unfortunately, the computational load of the additions can not be reduced analogously.

On the right side of Table 1 the individual timings for the needed operations are given. The complex multiplications and additions take most of the computing time, while the FFTs, due to the reduced number, are almost negligible. When calculating the gradient in (8) by using

$$\nabla_{\mathbf{h}_i} = \mathrm{Re}\left(\mathrm{IFFT}\left[\sum_{k=1}^{N_m}\boldsymbol{C}_{ki}^*\odot\mathrm{FFT}[\boldsymbol{\zeta}_k, L_n]\right]\right) \tag{13}$$

in part (c), the situation is almost the same, as the only difference is the complex transpose of $\boldsymbol{C}_{ki}$ and its different indexing scheme. These two calculations, consisting of simple multiplications and additions, are responsible for about 78% of the total computation time, as shown in Table 1 in the rows marked by "M+A", and therefore should be the primary target for optimization.

The calculation of $\boldsymbol{\zeta}_k$ using (9) in part (b), which represents the essential part of the algorithm, can actually be computed quite

Table 2: Comparison of the duration of the calculation of the single steps of one iteration of the algorithm using an implementation in CUDA. Problem size: 13×57. Times are given in ms. The last row shows the speedup in comparison to the Matlab implementation from Table 1.

Calc	Time	%	Step	Time	%	S-Up
(a) Eq. (1)	1.010	38.5	FFT	0.061	2.3	23
			M+A	0.728	27.8	188
			IFFT	0.221	8.4	31
(b) Eq. (9)	0.563	21.4		0.563	21.4	102
(c) Eq. (8)	1.038	39.6	FFT	0.221	8.4	31
			M+A	0.756	28.9	184
			IFFT	0.061	2.3	23
(d) Eq. (7)	0.013	0.5		0.013	0.5	15
Total	**2.626**	100.0				**133**

fast. It only takes about 16% of the total time. This is the secondary target for optimization.

The last steps consist of the calculation of the step size and making the gradient descent using (7). These parts are computationally negligible.

4. CUDA IMPLEMENTATION

In order to speed up the computation, we propose to offload the calculations to a CUDA enabled graphics hardware [10]. Due to the small stepsize $\mu(l)$ and high range of the coefficients of a RIR, double precision calculations are needed. Therefore, we have to use a hardware with at least 1.3 compute capability [10]. We have chosen to make our implementation on a TESLA M2050 card, which is, at the moment, the fastest available CUDA hardware with 512 double precision units.

For the calculation of FFT and IFFT the `cufft`-library, included in the CUDA framework, can be used. Although it is highly optimized for the target architecture and the implementation uses the batch mode, there is only a 20 to 30 times speedup compared to the CPU implementation, as shown in Table 2. This is mainly due to the fact that the used FFT length of 8192 is too small in order to fully utilize the processing power of CUDA hardware. But since all FFT operations combined are responsible for only about 21% of the running time, this is not a problem in this context.

The multiplication and summation in the frequency domain in (12), which uses most of the processing power, can be implemented very efficiently in one single kernel [11]. This is based on the fact that the computations for all frequencies and all filters are independent of each other, which perfectly corresponds to the CUDA principle of starting a huge amount of identical and simple threads [12]. In the previous example, this means we start $57 \times 4097 = 233529$ threads, and each of them calculates just 13 complex multiplications and additions. In this setup, all threads have the same length and consecutive threads access also consecutive memory addresses. This simple structure allows for a full coalesced memory access [12], which is essential for achieving the best performance by memory bounded operations. Exact measurements show that our implementation runs at about 124 GB/s, which is the peak memory bandwidth for the Tesla card with ECC memory turned on.

The kernel for calculating the multiplication and additions in (13) is implemented in an analogous way. Here, we start $13 \times 4097 = 53261$ threads which compute 57 complex multiplications and additions. The complex conjugations do not affect the speed, as this kernel is also memory bandwidth and not computationally limited. In both cases, we achieve an over 180 times speed up, as shown in Table 2. As these two kernels dominate the computational load, this is the main contribution for the acceleration of the whole implementation.

The calculations for (9), which represent the main part of the algorithm, can be split into two parts, which are then implemented by two separate kernels. The first part, the calculation of $\boldsymbol{b}_{uk}$ and accordingly $\boldsymbol{b}_{dk}$ in (10) and (11), can be performed by just one kernel that is invoked twice. As all computations are again independent of each other, we start one thread for every element of $\boldsymbol{b}_k$. This kernel consists of four simple operations (two multiplications, absolute value and a sign function) and one computationally expensive power operation. This is due to the fact that in current CUDA hardware, there is only direct support for single precision power operation, and the double precision calculation has to be emulated internally by using some more complex routines. For integer exponents, a small additional acceleration can be achieved via repeated multiplications instead of the power operation. Additionally, we also calculate and save $|g_{uk}|^{p_u} = |g_{uk}|^{p_u - 1} \cdot |g_{uk}|$ at a cost of one extra multiplication, as it allows for a very simple implementation of the second part of (9). Here, we need to calculate a norm without the final power operation, and with already calculated $|g_{uk}|^{p_u}$ this reduces to a simple addition of all elements. For this summation, we use a very efficient reduction scheme as in [11, 12]. Using all these optimizations, even with the expensive power operation, the CUDA implementation is about 100 times faster than the CPU version.

The different speedup factors for the different parts result in a 130-fold reduction of overall computation time, which allows for a near real-time operation for smaller setups.

The source code is available at [13].

5. SIMULATIONS

In the following simulations we tested the performance of the implementation for different sized setups. In all cases, we used RIRs of length $L_c = 2000$ taps, prefilters with $L_h = 3000$ taps, which resulted in overall RIRs of length 4999 taps.

In Table 3 the results for different sized setups are summarized. For small ones, convergence could be achieved after 2500 iterations. With the CUDA implementation a real-time operation can be achieved, as the total computational time is about only one second. But for theses small cases, the overall speedup, compared to the CPU implementation, is only 30 fold. In this case, there could not be enough threads started to fully utilize the processing power of the CUDA hardware. Furthermore, the slow FFT operations use a relatively large part of the computation time and slow down the whole process. In Figure 1 the speedup for one iteration in dependency on the problem size $N_m \cdot N_s$ is shown.

For bigger setups, more iterations are needed for convergence. For the 13×57 setup using 10^4 iterations an acceptable solution can be found, but for an optimum result 10^5 iterations are needed. In Figure 2 the results are presented. Figure 2 (a) depicts the RIR magnitudes $|g_i(n)|$ without any preprocessing. The black line is the average temporal masking curve. The reshaped RIRs $g_i(n)$ in Figure 2 (b) are under the the average temporal masking curve and should not cause perceivable echoes. In Figure 2 (c) the average RIR of 30 random points in the listening area is shown. Here almost all taps are under the average temporal masking curve and, usually, only small reverberation should be perceived.

Table 3: Total runtime for different configurations.

Setup	Iterations	Matlab	Cuda	Speed Up
2×4	2500	26.9 s	**0.88 s**	30
4×8	2500	61.8 s	**1.06 s**	58
6×16	5000	4.6 min	3.65 s	77
10×32	10000	24.3 min	14.30 s	101
13×57	10000	58.5 min	26.26 s	133
13×57	100000	9.7 h	4.37 min	133

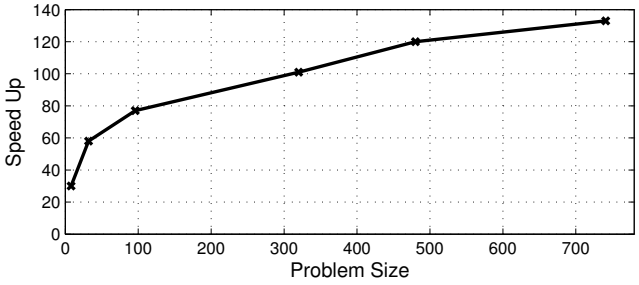

Figure 1: The speedup of the CUDA implementation depending on the problem size $N_m \cdot N_s$.

6. SUMMARY

Multichannel room impulse response reshaping algorithms are able to equalize a listening area. Unfortunately, the computational costs are too high for a quick real-time design of the optimal filters. In this work, we studied a CUDA implementation which is able to speed up these calculation by a factor of 130. The new implementation is able to perform near real-time. The algorithm has been tested on different configurations.

7. REFERENCES

[1] S. T. Neely and J. B. Allen, "Invertibility of a room impulse response," *J. Acoustical Society of America*, vol. 68, pp. 165–169, July 1979.

[2] B. D. Radlovic and R. A. Kennedy, "Nonminimum-phase equalization and its subjective importance in room acoustics," *IEEE Transactions on Speech and Audio Processing*, vol. 8, no. 6, pp. 728–737, Nov. 2000.

[3] M. Kallinger and A. Mertins, "Room impulse response shortening by channel shortening concepts," in *Proc. Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA , USA, Oct. 30 - Nov. 2 2005, pp. 898–902.

[4] A. Mertins, T. Mei, and M. Kallinger, "Room impulse response shortening/reshaping with infinity- and p-norm optimization," *IEEE Trans. Audio, Speech, and Language Processing*, vol. 18, no. 2, pp. 249–259, Feb. 2010.

[5] T. Mei, A. Mertins, and M. Kallinger, "Room impulse response reshaping/shortening based on least mean squares optimization with infinity norm constraint," in *Proc. 16th International Conference on Digital Signal Processing*, July 5–7, 2009, pp. 1–6.

[6] J. Blauert, J. Mourjopoulos, and J. Buchholz, "Room masking: Understanding and modelling the masking of reflections in rooms," in *Audio Engineering Society Convention 110*, 5 2001.

[7] L. D. Fielder, "Practical limits for room equalization," in *Audio Engineering Society Convention 111*, 11 2001, pp. 1–19.

[8] T. Ajdler, L. Sbaiz, and M. Vetterli, "The plenacoustic function and its sampling," *Signal Processing, IEEE Transactions on*, vol. 54, no. 10, pp. 3790 –3804, oct. 2006.

[9] T. Mei and A. Mertins, "On the robustness of room impulse response reshaping," in *Proc. International Workshop on Acoustic Echo and Noise control (IWAENC)*, Tel Aviv, Israel, Aug. 2010.

[10] http://www.nvidia.com/object/cuda_home_new.html.

[11] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 7 2010.

[12] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*, 1st ed. Morgan Kaufmann, 2 2010.

[13] http://www.isip.uni-luebeck.de/index.php?id=610.

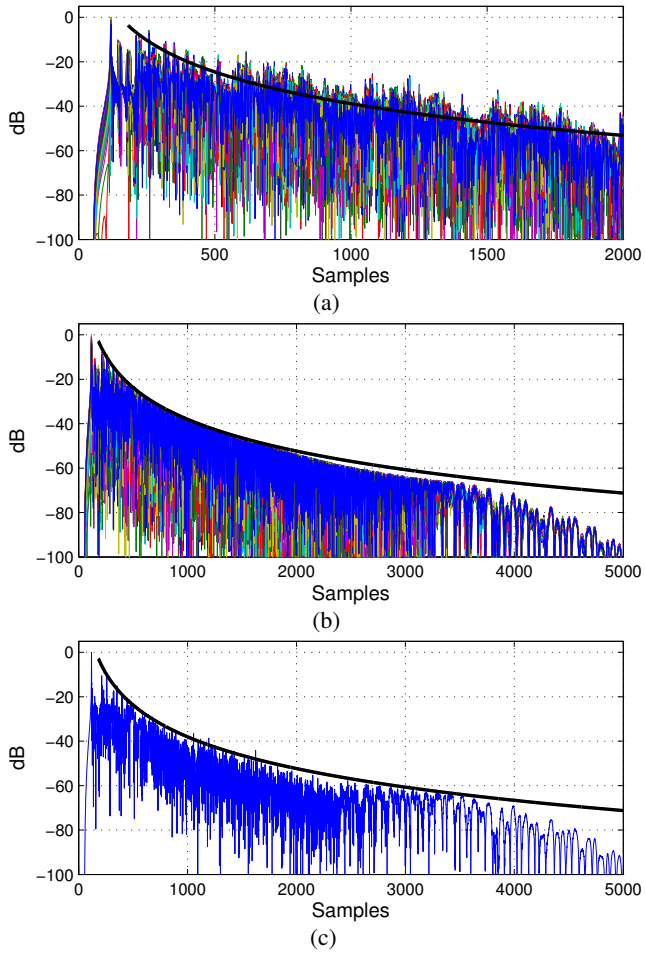

Figure 2: RIR magnitudes and spatial reshaping: (a) $|g_i(n)|$ without preprocessing; (b) $|g_i(n)|$ after convergence; (c) Averaged global RIR of 30 random points in the listening area. The black line is the average temporal masking curve.